NC STATE UNIVERSITY

Linked Lists

Reading
Zybooks: Chapter 14

CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty

2

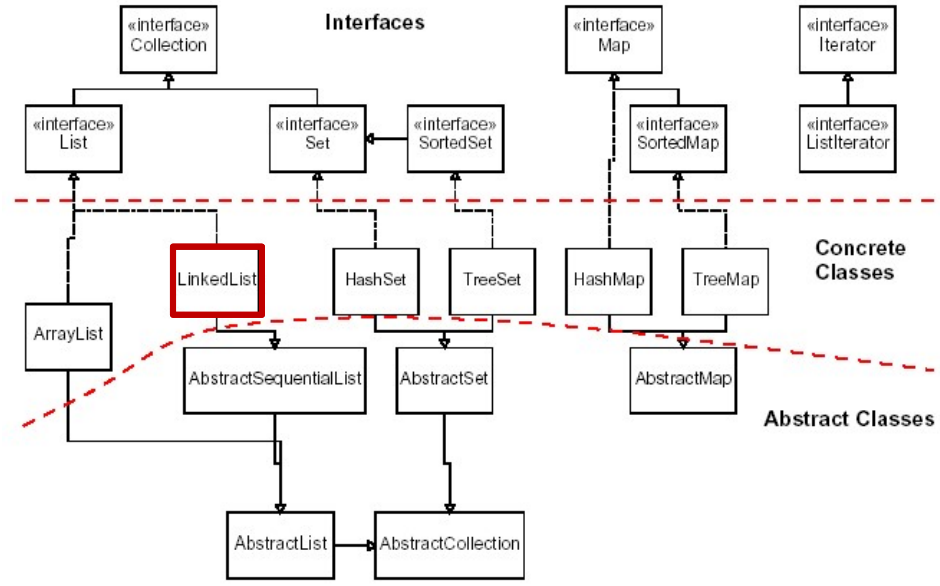
NC STATE UNIVERSITY

Collections

- Data structures – stores elements in a manner that makes it easy for a client to work with the elements
- Specific collections are specialized for particular types of operations
  - Ordering
  - Duplicates
  - Add/remove
- Common set of operations: add, remove, get, contains, size

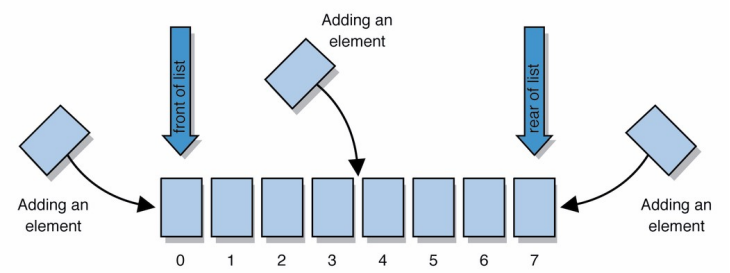
CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty 3

3



Lists

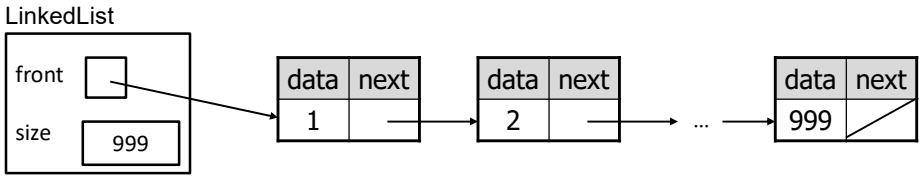
- List: an ordered collection of elements
  - Accessible by a zero-based index
  - Size: number of elements in the list
  - Elements may be added to *an empty list*, at the *front*, *middle*, and *back*



NC STATE UNIVERSITY

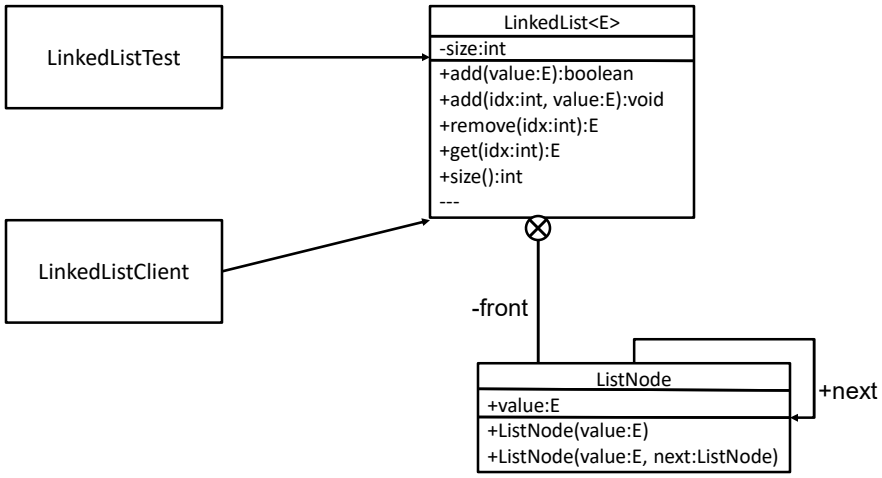
`LinkedList` Class

- Let's write a collection library class named **`LinkedList`**.
  - Has standard list methods
    - **`add`**, **`add`**, **`get`**, **`indexOf`**, **`remove`**, **`size`**, **`toString`**
  - The list is internally implemented as linked nodes
    - The **`LinkedList`** has a reference to its **`front`** as a field
    - **`null`** represents the end of the list; a **`null`** front signifies an empty list



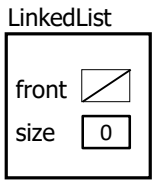
NC STATE UNIVERSITY

`LinkedList` Class Diagram



LinkedList

```
public class LinkedList<E> {
   private ListNode front;
   private int size;

   //Standard list methods go here

   private class ListNode {
      public E data;
      public ListNode next;
      public ListNode(E data) { this(data, null); }
      public ListNode(E data, ListNode next) {
         this.data = data;
         this.next = next;
      }
   }
}
```



Testing Lists

- Test each List method
  - Empty list
  - Front of the list
  - Middle of the list
  - Back of the list
- After each operation
  - Length is correct
  - Contents are correct (are the elements in the right order?)
- Then test the methods in combination
  - maybe `add` usually works, but fails after you call `remove`
  - what happens if I call `add` then `size`? `remove` then `toString`?
  - make multiple calls; maybe `size` fails the second time only

NC STATE UNIVERSITY

LinkedList()

- Start by testing the constructor
  - Should create an empty list of elements

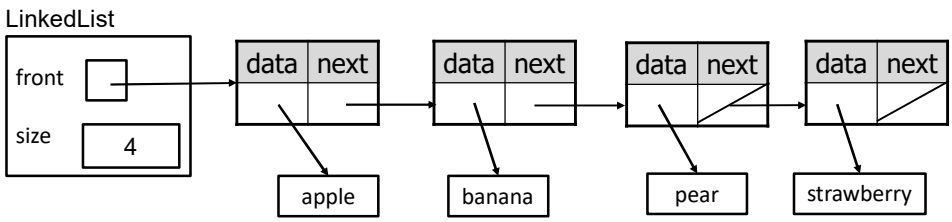
```
public class LinkedListTest {
    @Test
    public void testLinkedList() {
        LinkedList<String> l = new LinkedList<String>();
        assertEquals(0, l.size());
    }
}
```


NC STATE UNIVERSITY

LinkedList.add(value)

- Test adding elements to the end of the list
  - Check size AND that elements are in the right order after the add

```
list.add("strawberry");
```



Implementing `LinkedList.add(value)`

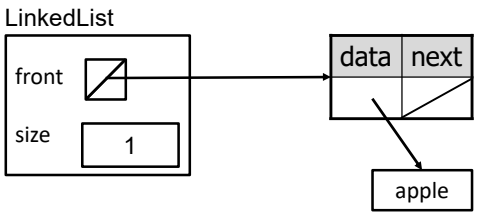
```
// Adds the given value to the end of the list.
public void add(E value) {
    ...
}
```

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?

Implementing `LinkedList.add(value)` – Scenario 1

- Scenario 1: Adding to an empty list

  `list.add("apple");`



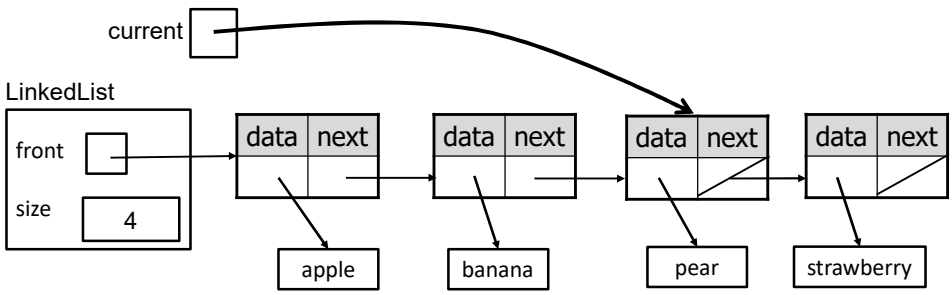
`LinkedList.add(value)` – Scenario 1

```
// Adds the given value to the end of the list.
public void add(E value) {
    if (front == null) {
        // adding to an empty list - special case updating front
        front = new ListNode(value);
    } else {
        // adding to end of an existing list

        ...

    }
    size++; // don't forget to increment size
}
```


Implementing `LinkedList.add(value)` – Scenario 2

Scenario 2: Adding to a list with contents

```
list.add("strawberry");
```



- **Don't fall off the edge!!!**

NC STATE UNIVERSITY

`LinkedList.add(value)` – Complete

```
// Adds the given value to the end of the list.
public void add(E value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to end of an existing list
        ListNode current = front;
        while (current.next != null) { // key concept - look ahead!
            current = current.next;
        }
        // add new node
        current.next = new ListNode(value);
    }
    size++;
}
```


NC STATE UNIVERSITY

`LinkedList.get(idx)` and `LinkedList.size()`

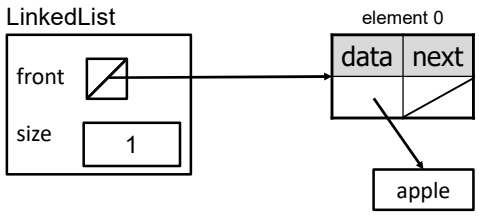
- Methods are needed to test other LinkedList methods
- **`get(idx)`**
  - Use a current reference to traverse (don't lose the global list)
  - Can use a for loop
- **`size()`**
  - Return the size field
  - But if we didn't have a size field, what is the algorithm?

`LinkedList.add(idx, value)` – Scenario 1

```
LinkedList<String> list = new LinkedList<String>();
```

- Scenario 1: Adding to an empty list

  `list.add(0, "apple");`

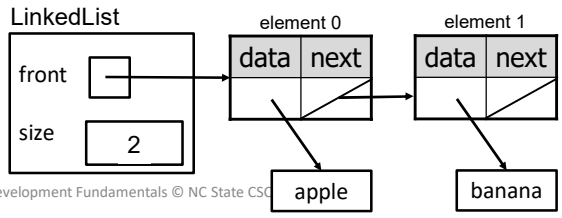


`LinkedList.add(idx, value)` – Scenario 2

```
LinkedList<String> list = new LinkedList<String>();
list.add(0, "apple");
```

- Scenario 2: Adding to the end of the list

  `list.add(1, "banana");`



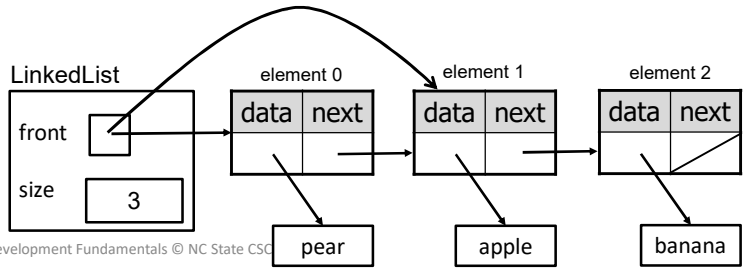
NC STATE UNIVERSITY

LinkedList.add(idx, value) – Scenario 3

```
LinkedList<String> list = new LinkedList<String>();
list.add(0, "apple");  //empty list
list.add(1, "banana"); //end of list
```

- Scenario 3: Adding to the front of the list

```
list.add(0, "pear");
```



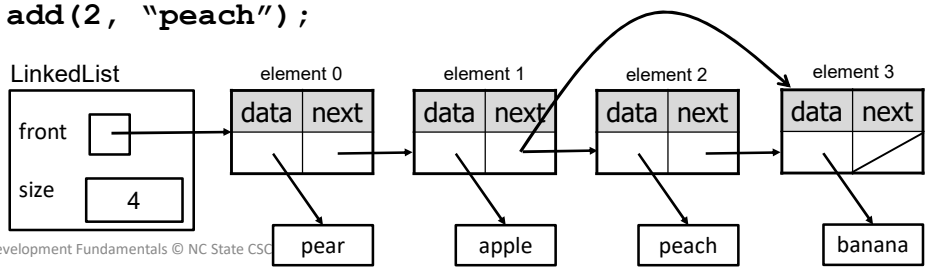
NC STATE UNIVERSITY

LinkedList.add(idx, value) – Scenario 4

```
LinkedList<String> list = new LinkedList<String>();
list.add(0, "apple");  //empty list
list.add(1, "banana"); //end of list
List.add(0, "pear");   //front of list
```

- Scenario 4: Adding to the middle of the list

```
list.add(2, "peach");
```



NC STATE UNIVERSITY

LinkedList.add(idx,value)

```
// Inserts the value at the index.
public void add(int idx, E value) {
    if (idx == 0) { // Special case: front

    } else {
        // Use current reference

        // Stop BEFORE idx to add at

        // Add the node so that it points to
        // the next
    }
    size++;
}
```


NC STATE UNIVERSITY

Conceptual Questions

- What is the difference between a `LinkedList` and a `ListNode`?
- What is the difference between an empty list and a `null` list?
  - How do you create each one?
- Why are the fields of `ListNode` public? Is this bad style?
- What effect does this code have on a `LinkedList`?

```
ListNode current = front;
current = null;
```

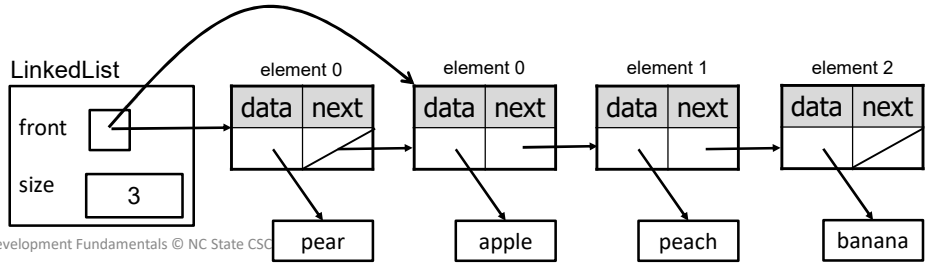
NC STATE UNIVERSITY

`LinkedList.remove(idx)` – Scenario 1

```
LinkedList<String> list = new LinkedList<String>();
list.add("pear"); list.add("apple"); list.add("peach"); list.add("banana");
```

- Scenario 1: Removing from the front

  `list.remove(0);`



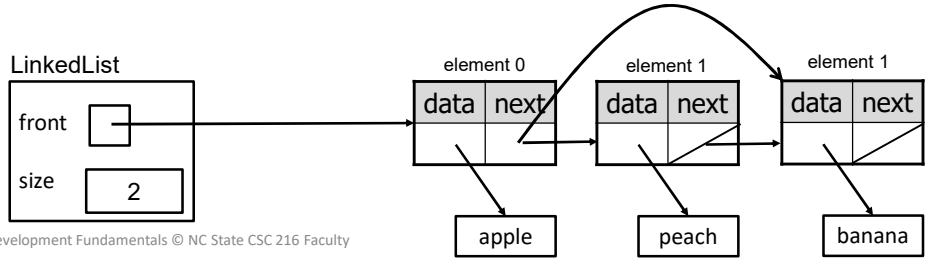
NC STATE UNIVERSITY

`LinkedList.remove(idx)` – Scenario 2

```
LinkedList<String> list = new LinkedList<String>();
list.add("pear"); list.add("apple"); list.add("peach"); list.add("banana");
list.remove(0); //remove from front
```

- Scenario 2: Removing from the middle

  `list.remove(1);`



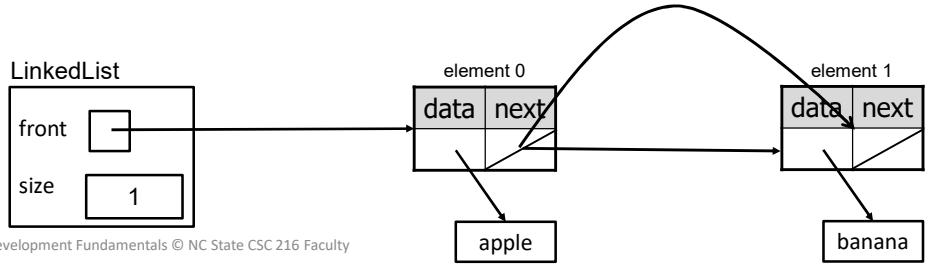
NC STATE UNIVERSITY

`LinkedList.remove(idx)` – Scenario 3

```
LinkedList<String> list = new LinkedList<String>();
list.add("pear"); list.add("apple"); list.add("peach"); list.add("banana");
list.remove(0); list.remove(1); //remove from front, middle
```

- Scenario 3: Removing from the end

  `list.remove(1);`



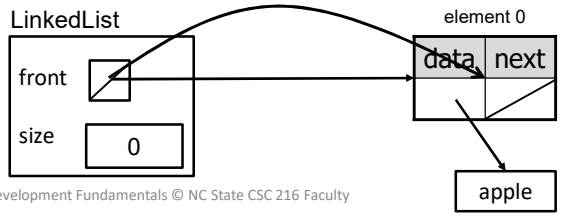
NC STATE UNIVERSITY

`LinkedList.remove(idx)` – Scenario 4

```
LinkedList<String> list = new LinkedList<String>();
list.add("pear"); list.add("apple"); list.add("peach"); list.add("banana");
list.remove(0); list.remove(1); list.remove(1); //front,mid,end
```

- Scenario 4: Removing the last element

  `list.remove(0);`



NC STATE UNIVERSITY

LinedList.remove(idx)

```
// Removes value at given index from list.
// Precondition: 0 <= idx < size()
public E remove(int idx) {
    E value = null;
    if (idx == 0) { // Special Case: front of list
        value = front.data;
        front = front.next;
    } else {
        // removing from elsewhere in the list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        value = current.next.data;
        current.next = current.next.next;
    }
    size--;
    return value;
}
```


30

NC STATE UNIVERSITY

LinkedList.addSorted(value)

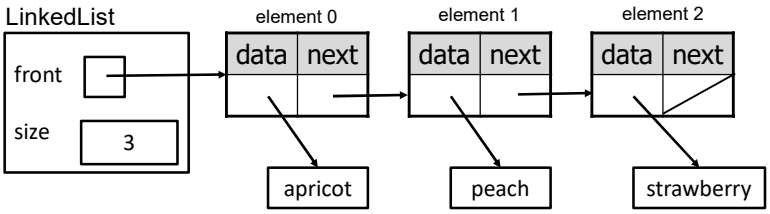
- Assuming a list is in sorted order, **`addSorted(value)`** will add a value to the list in sorted order.
- Let's explore how to implement **`addSorted(value)`**
- We'll explore the different cases…

31

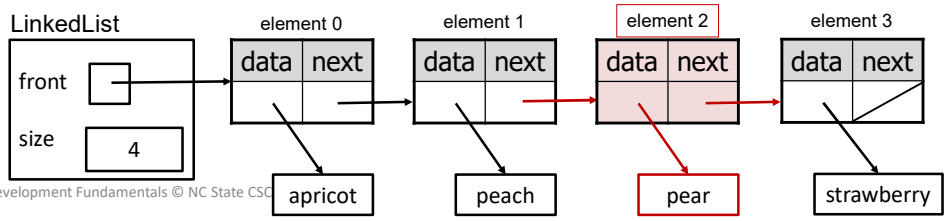
NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 1

Scenario 1: the common case of adding to the middle of the list



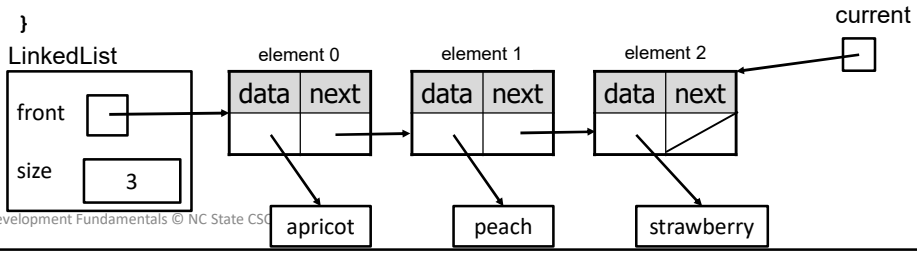

```
list.addSorted("pear");
```



NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 1 Incorrect

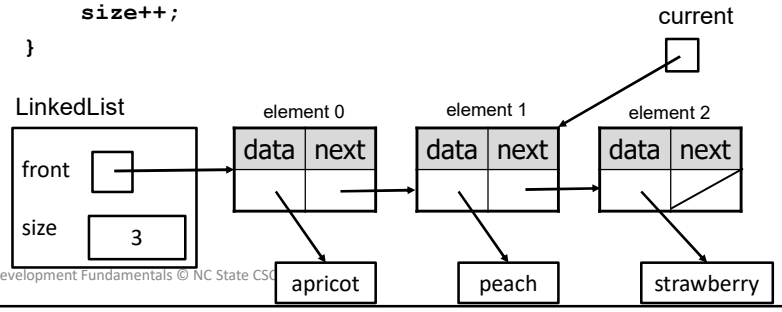
```
// Scenario 1: the common case of adding to the middle of the list
public void addSorted(E value) {
    ListNode current = front;
    while (current.data.compareTo(value) < 0) {
        current = current.next;
    }
    //where is current pointing when we're done?
    current.next = new ListNode(value, current.next);
    size++;
}
```



NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 1 Correct

```
// Scenario 1: the common case of adding to the middle of the list
public void addSorted(E value) {
    ListNode current = front;
    while (current.next.data.compareTo(value) < 0) { //Peek ahead!!!
        current = current.next;
    }
    current.next = new ListNode(value, current.next);
    size++;
}
```

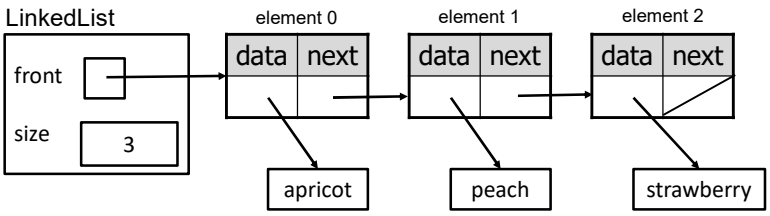


34

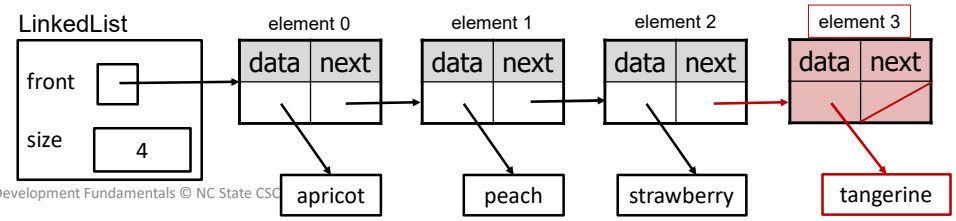
NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 2

Scenario 2: adding to the end of the list




```
list.addSorted("tangerine");
```

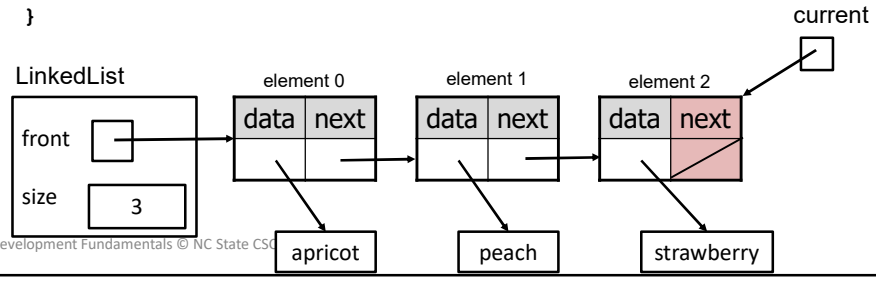


35

NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 2 Incorrect

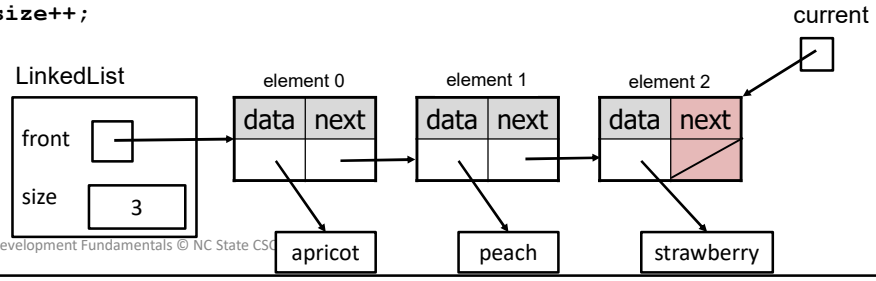
```
// Scenario 2: adding to the end of the list
public void addSorted(E value) {
    ListNode current = front;
    while (current.next.data.compareTo(value) < 0) { //NPE!!!
        current = current.next;
    }
    current.next = new ListNode(value, current.next);
    size++;
}
```



NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 2 Correct

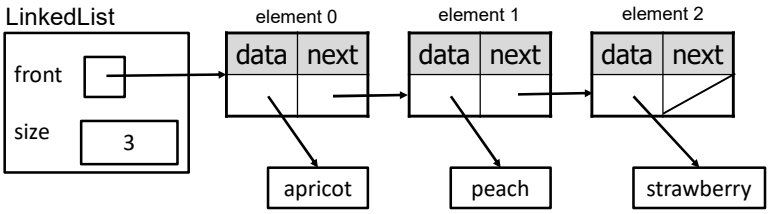
```
// Scenario 2: adding to the end of the list
public void addSorted(E value) {
    ListNode current = front;
    // check for end of the list before peeking ahead
    while (current.next != null && current.next.data.compareTo(value) < 0) {
        current = current.next;
    }
    current.next = new ListNode(value, current.next);
    size++;
}
```



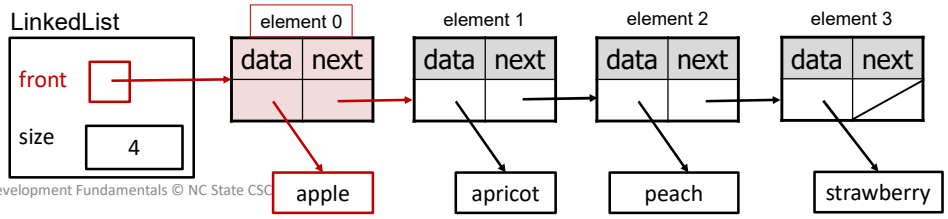
NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 3

Scenario 3: adding to the front of the list



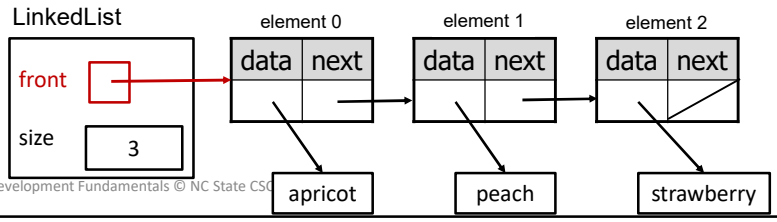

```
list.addSorted("apple");
```



NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 3 Correct

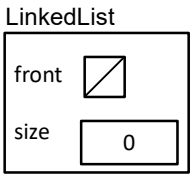
```
// Scenario 3: adding to the front of the list
public void addSorted(E value) {
    if (value.compareTo(front.data) < 0) { //add to the front
        front = new ListNode(value, front);
    } else {
        // insert into middle or end of list
    }
    size++;
}
```



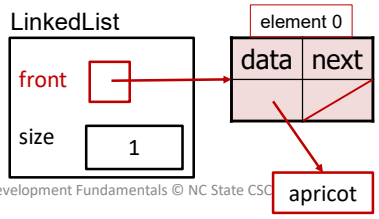
NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 4

Scenario 4: adding an element to an empty list




```
list.addSorted("apricot");
```

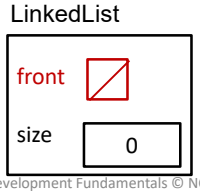


NC STATE UNIVERSITY

`LinkedList.addSorted(value)` – Scenario 4 Incorrect

Scenario 4: adding an element to an empty list

```
public void addSorted(E value) {
    if (value.compareTo(front.data) < 0) { //add to the front - NPE!!!!
        front = new ListNode(value, front);
    } else {
        // insert into middle or end of list
    }
    size++;
}
```



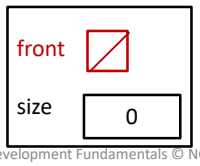
NC STATE UNIVERSITY

LinkedList.addSorted(value) – Scenario 4 Correct

Scenario 4: adding an element to an empty list

```
public void addSorted(E value) {
    if (front == null || value.compareTo(front.data) < 0) {
                //add to the front – check for empty list
        front = new ListNode(value, front);
    } else {
        // insert into middle or end of list
    }
    size++;
}
```

LinkedList



NC STATE UNIVERSITY

LinkedList.addSorted(value) – Complete

```
public void addSorted(E value) {
   // Add to front of a list, including an empty list
   if (front == null || value.compareTo(front.data) < 0) {
      front = new ListNode(value, front);
   } else { // insert into middle or end of list
      ListNode current = front;
      while (current.next != null && current.next.data.compareTo(value) < 0) {
         current = current.next;
      }
      current.next = new ListNode(value, current.next);
   }
   size++;
}
```

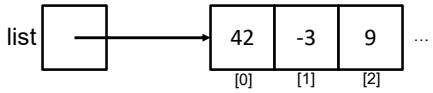
Scenario 1: Middle of List | Scenario 2: End of List

Scenario 3: Front of List | Scenario 4: Empty List

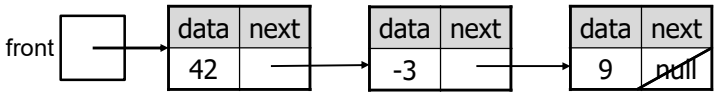
NC STATE UNIVERSITY

Linear Data Structures

- We can implement lists with two underlying data structures:
  - an **array** of all elements
    - `ArrayList`, `Stack`, `HashSet`, `HashMap`



  - a set of **linked objects**, each storing one element, and one or more reference(s) to other element(s)
    - `LinkedList`, `TreeSet`, `TreeMap`



NC STATE UNIVERSITY

A `List` Interface

```
// Represents a list of generic objects.
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> { ... }
public class LinkedList<E> implements List<E> { ... }
```